UFR IM²AG

UNIVERSITÉ Grenoble Alpes

UE INF404 - Projet Logiciel

Calculatrice : étape 4 (fin !)

Evaluation d'une expression arithmétique générale

L2 Informatique

Année 2022 - 2023

Rappel des précédents épisodes (1)

Ecrire un interpréteur d'expressions arithmétiques

Version 1 = "Expressions Arithmétiques Simples" (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Version 2 = "Expressions Arithmétiques Entièrement Parenthésées" (EAEP)

- les opérandes sont des entiers
- parenthèses obligatoires autour de chaque opération
- évaluation selon le parenthésage ...

Rappel des précédents épisodes (1)

Ecrire un interpréteur d'expressions arithmétiques

Version 1 = "Expressions Arithmétiques Simples" (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Version 2 = "Expressions Arithmétiques Entièrement Parenthésées" (EAEP)

- les opérandes sont des entiers
- parenthèses obligatoires autour de chaque opération
- évaluation selon le parenthésage . . .

Rappel des précédents épisodes (1)

Ecrire un interpréteur d'expressions arithmétiques

Version 1 = "Expressions Arithmétiques Simples" (EAS)

- les opérandes sont des entiers
- opérateurs arithmétiques usuels (+, -, *, /)
- pas de priorités (évaluation de gauche à droite)

Version 2 = "Expressions Arithmétiques Entièrement Parenthésées" (EAEP)

- les opérandes sont des entiers
- parenthèses obligatoires autour de chaque opération
- évaluation selon le parenthésage ...

Rappel des précédents épisodes (2)

"Expressions Arithmétiques Générales" (EAG)

- parenthésage lorsque nécessaire
- évaluation selon les règles de priorités usuelles ...

Programmation ?

- syntaxe spécifiée par une grammaire "structurée"
- analyse syntaxique =
  procédures **récursives**, guidée par la grammaire (c.f. TP3)

La suite = évaluation d'une EAG ?

Rappel des précédents épisodes (2)

"Expressions Arithmétiques Générales" (EAG)

- parenthésage lorsque nécessaire
- évaluation selon les règles de priorités usuelles ...

Programmation ?

- syntaxe spécifiée par une grammaire "structurée"
- analyse syntaxique =
  procédures **récursives**, guidée par la grammaire (c.f. TP3)

La suite = évaluation d'une EAG ?

Grammaire (structurée) des EAG

expression = séquence de termes séparés par PLUS, MOINS
terme = séquence de facteurs séparés par MUL, DIV
facteur = ENTIER ou expressions parenthésées ...

$$
\begin{aligned}
\textit{eag} &\rightarrow \textit{seq\_terme} \\
\textit{seq\_terme} &\rightarrow \textit{terme}\ \textit{suite\_seq\_terme} \\
\textit{suite\_seq\_terme} &\rightarrow \textit{op1}\ \textit{terme}\ \textit{suite\_seq\_terme} \\
\textit{suite\_seq\_terme} &\rightarrow \varepsilon \\
\textit{terme} &\rightarrow \textit{seq\_facteur} \\
\textit{seq\_facteur} &\rightarrow \textit{facteur}\ \textit{suite\_seq\_facteur} \\
\textit{suite\_seq\_facteur} &\rightarrow \textit{op2}\ \textit{facteur}\ \textit{suite\_seq\_facteur} \\
\textit{suite\_seq\_facteur} &\rightarrow \varepsilon \\
\textit{facteur} &\rightarrow \textit{ENTIER} \\
\textit{facteur} &\rightarrow \texttt{PARO}\ \textit{eag}\ \texttt{PARF} \\
\textit{op1} &\rightarrow \texttt{PLUS} \\
\textit{op1} &\rightarrow \texttt{MOINS} \\
\textit{op2} &\rightarrow \texttt{MUL}
\end{aligned}
$$

Arbres de dérivation

Soit $G = (V_t, V_n, Z, P)$ une grammaire et w un mot de $L(G)$.

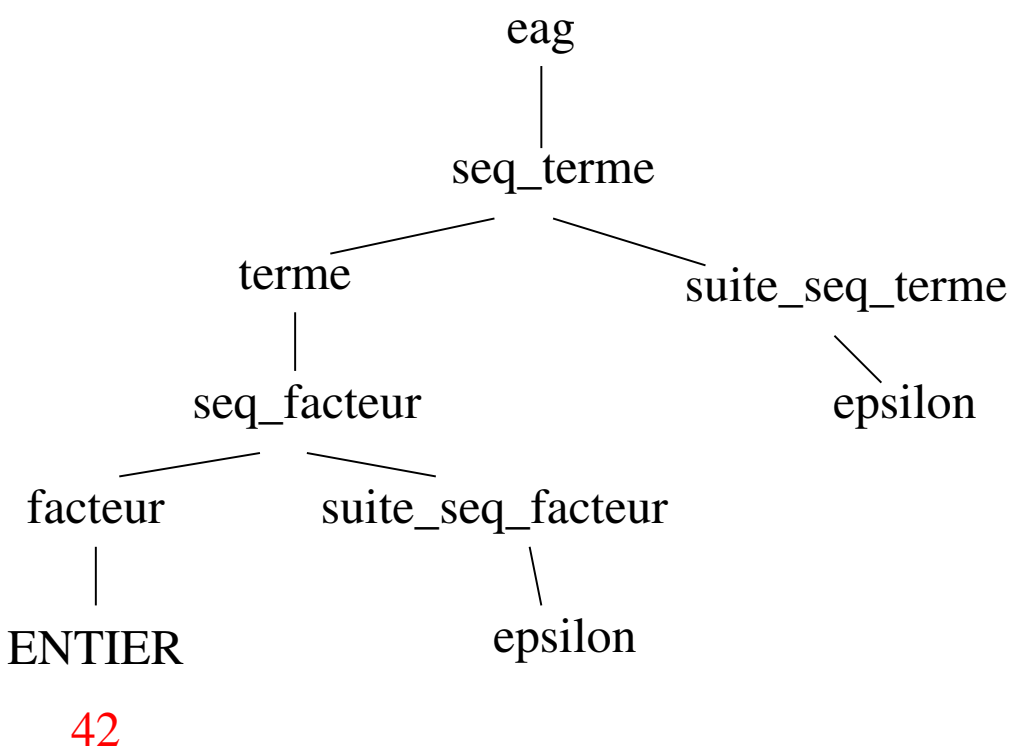
On appelle arbre de dérivation (ou arbre syntaxique) de w dans G tout arbre n-aire A dont les noeuds sont des éléments de $(V_t \cup V_n)$ et tel que :

- la racine de A est Z (l'axiome de G) ;
- les feuilles de A sont des éléments de $V_t \cup \{\varepsilon\}$ et la séquence gauche-droite des feuilles de A est égale à w ;
- les noeuds non feuilles de A sont des éléments de V_n et si un noeud non feuille X a pour fils u_1, u_2, $\ldots u_n$ dans A alors la règle $X \rightarrow u_1.u_2 \ldots u_n$ doit appartenir à P.

A tout mot de $L(G)$ on peut associer un arbre de dérivation.
Si cet arbre est **unique** alors la grammaire est dite **non ambigüe**.

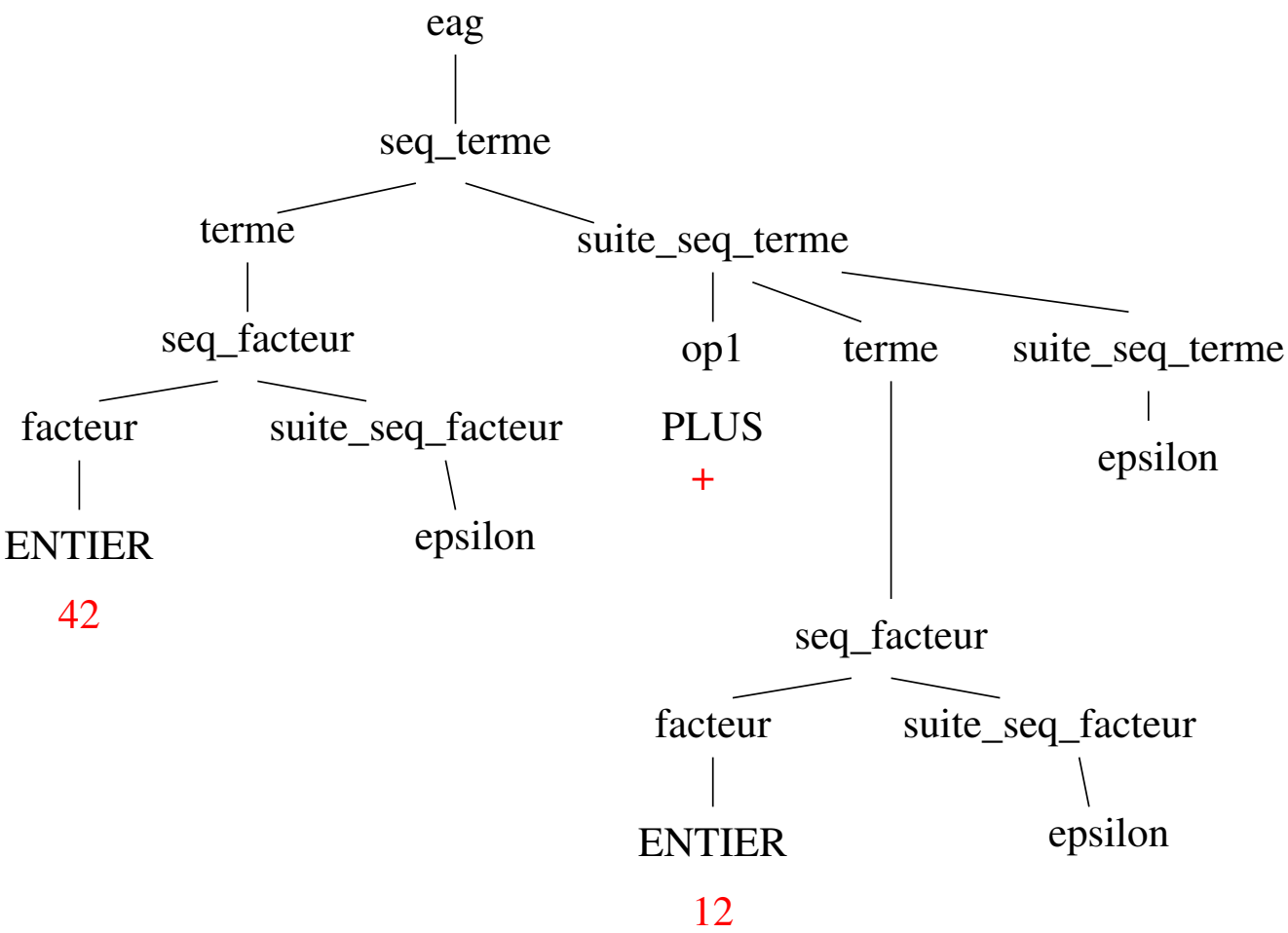
Exemple d'arbre de dérivation (1)

42



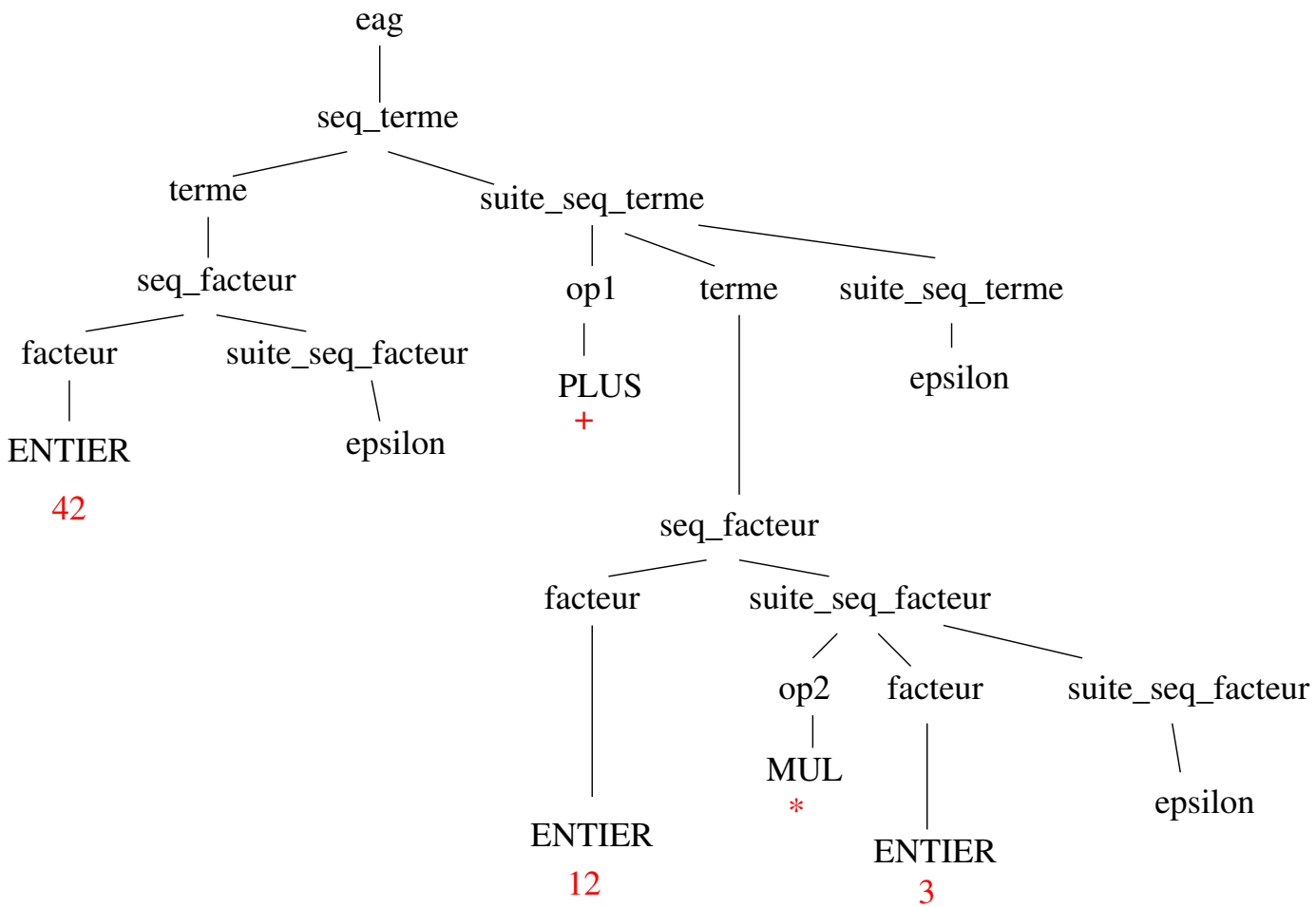
Exemple d'arbre de dérivation (2)

42 + 12



Exemple d'arbre de dérivation (3)

42 + 12 * 3



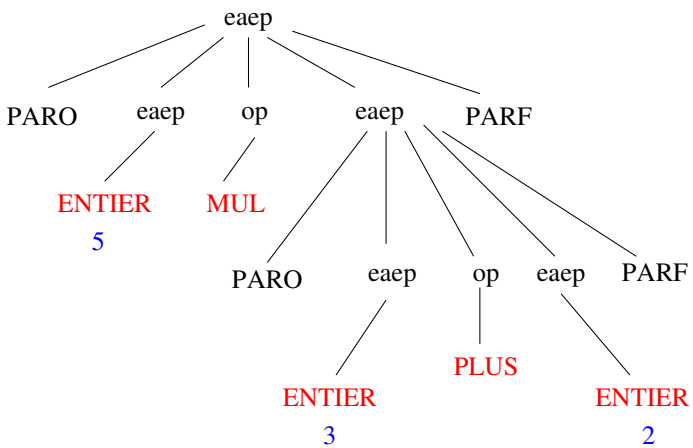
Arbre abstrait (Abstract Syntax Tree, AsT)

arbre de dérivation $\sim$ exécution de l'analyseur syntxique
(arbre des appels des procédures récursives "extraites" de la grammaire)

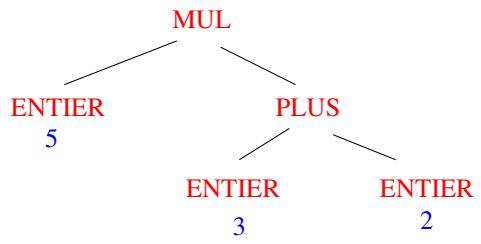
Arbre abstrait = Une "simplification" de l'arbre de dérivation
$\rightarrow$ permet de représenter les "informations utiles"
(ex : structure du texte lu, éléments importants)

$\Rightarrow$ **représentation intermédiaire** utilisée en sortie de l'analyseur

Exemple (sur les EAEP) : `(5 * (3 + 2))`



Arbre de Derivation



Arbre Abstrait

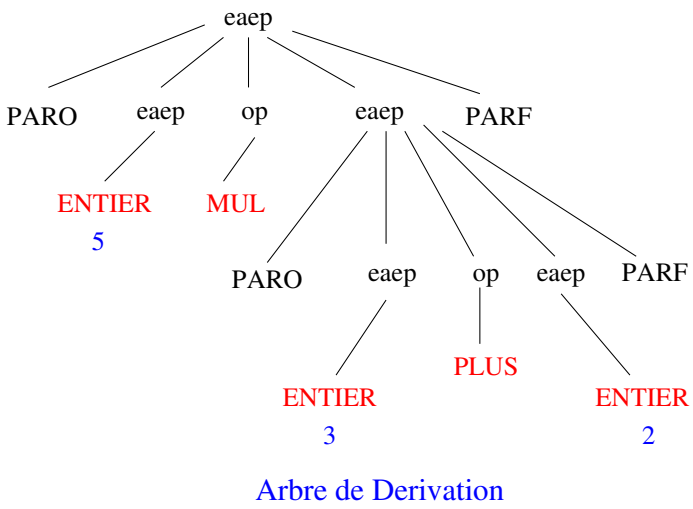
Arbre abstrait (Abstract Syntax Tree, AsT)

arbre de dérivation $\sim$ exécution de l'analyseur syntxique
(arbre des appels des procédures récursives "extraites" de la grammaire)

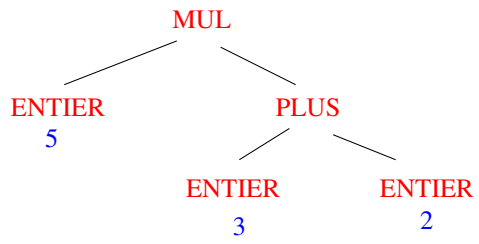
Arbre abstrait = Une "simplification" de l'arbre de dérivation
$\rightarrow$ permet de représenter les "informations utiles"
(ex : structure du texte lu, éléments importants)

$\Rightarrow$ **représentation intermédiaire** utilisée en sortie de l'analyseur

Exemple (sur les EAEP) : `(5 * (3 + 2))`



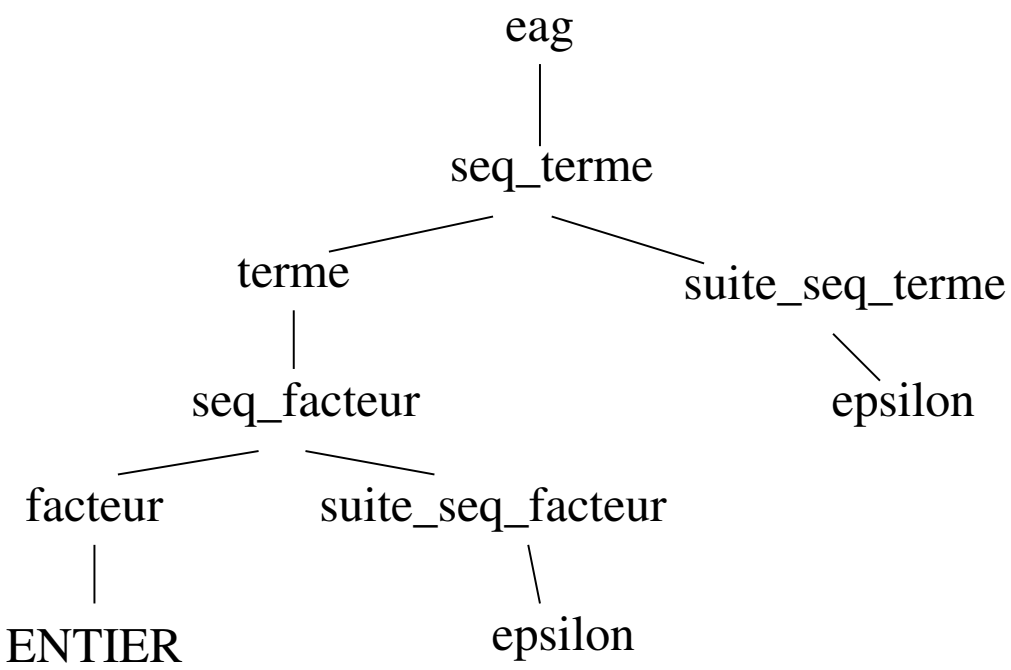
Arbre de Derivation



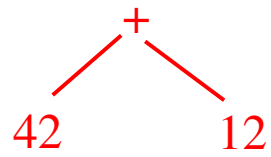
Arbre Abstrait

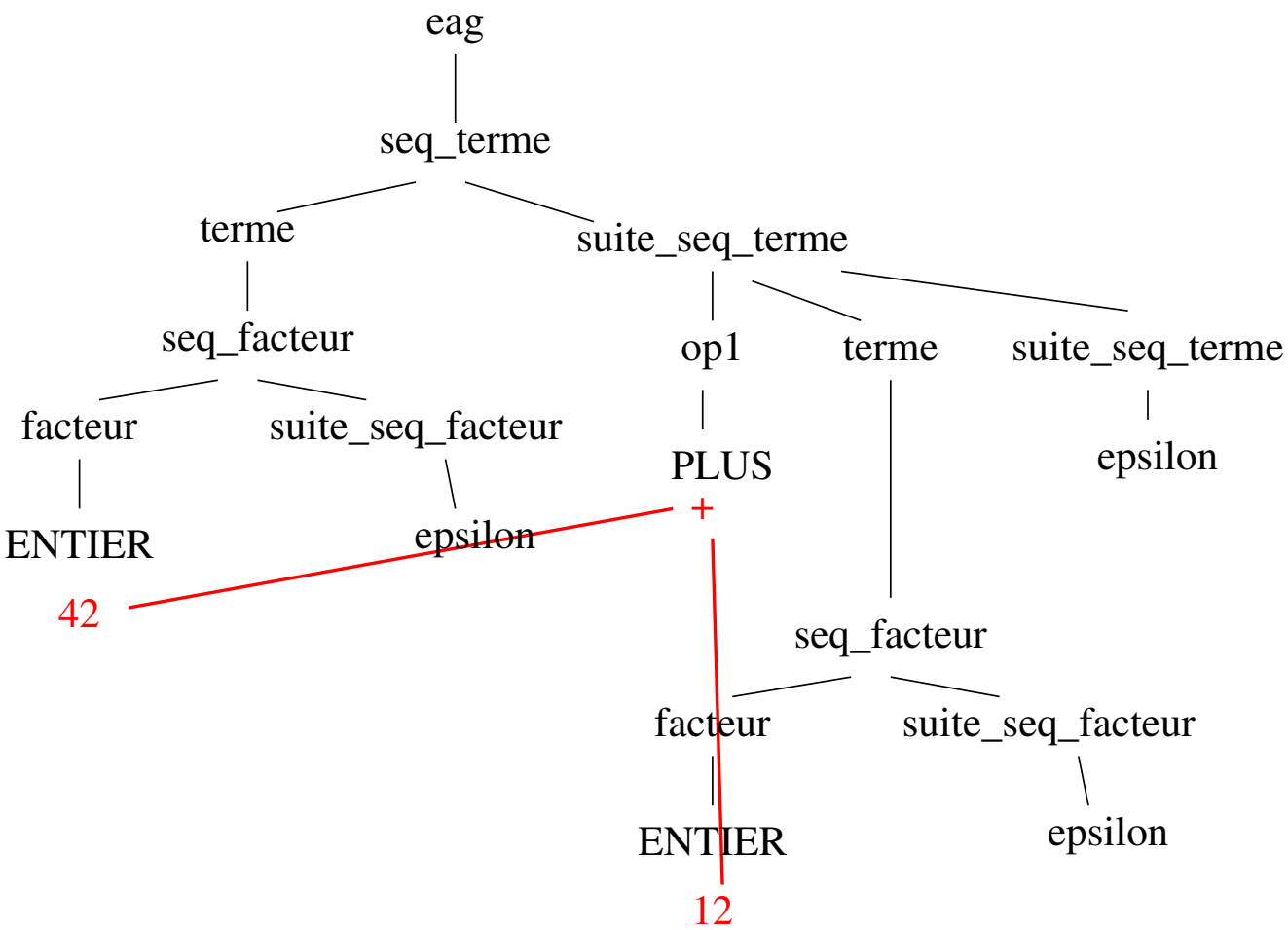
Exemple d'arbre abstrait : 42

42

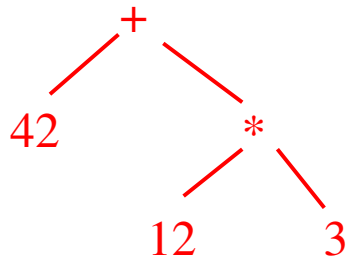


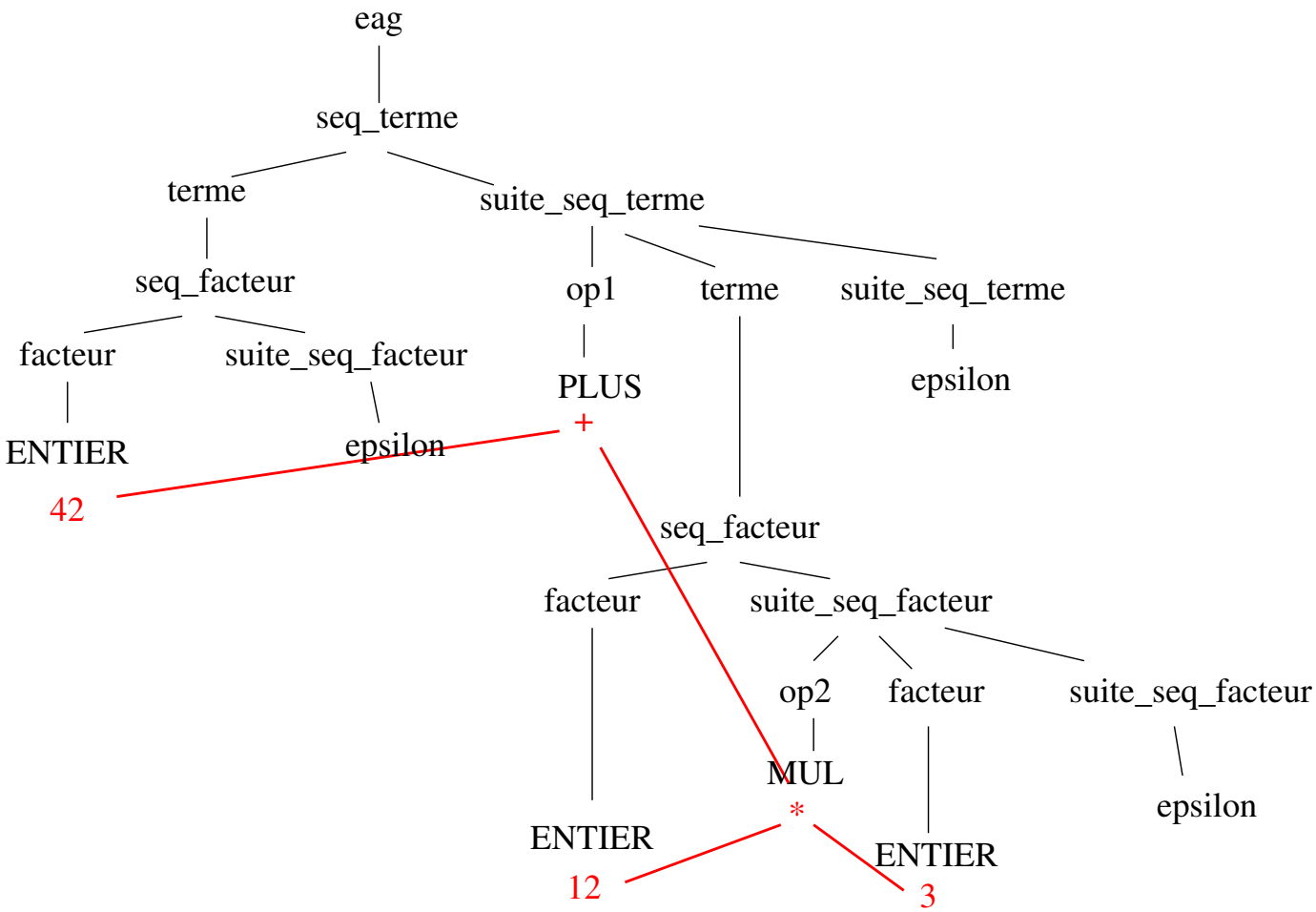
Exemple d'arbre abstrait : 42 + 12





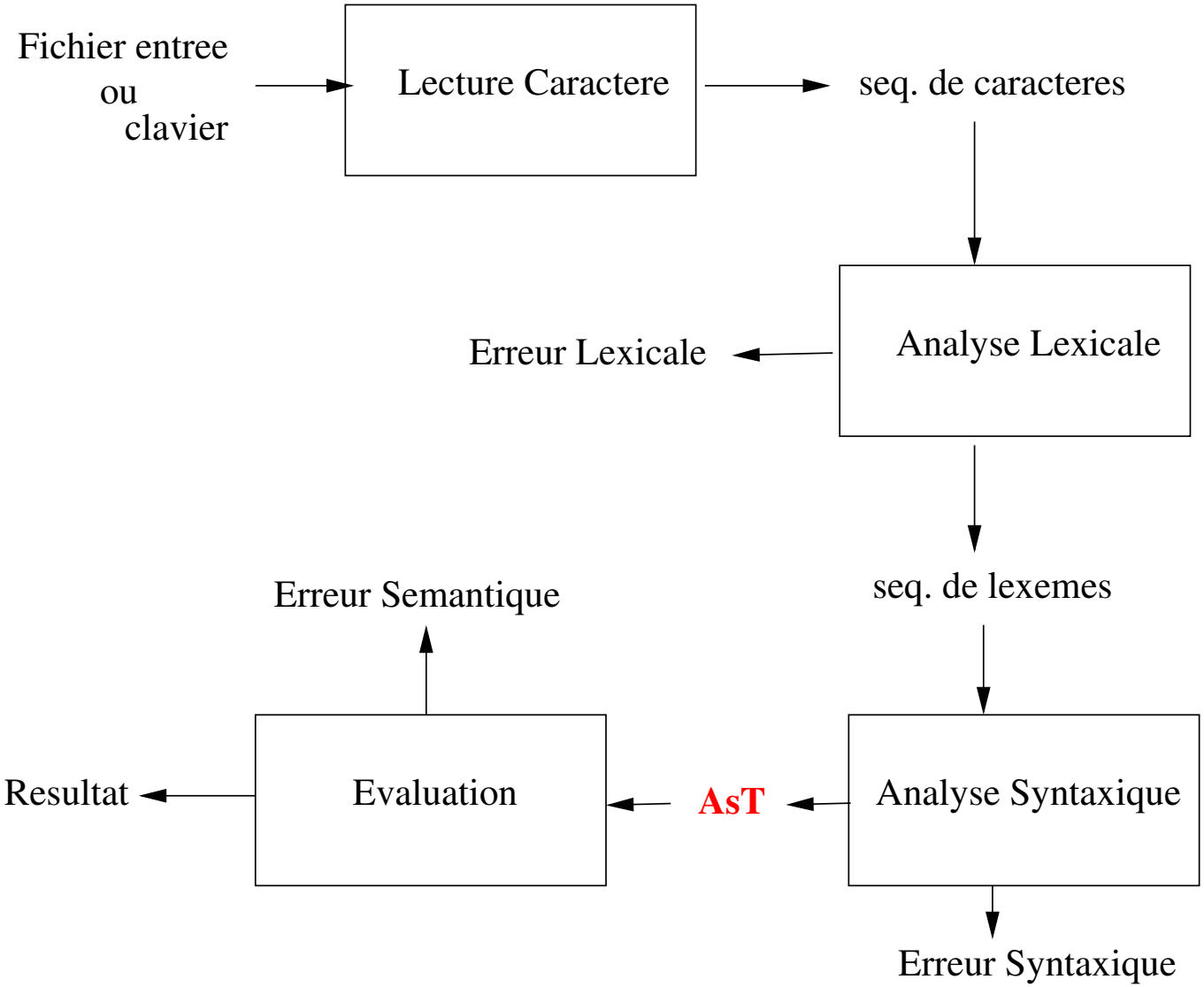
Exemple d'arbre abstrait : 42 + 12*3





Structure de la calculatrice

Quatre composants/modules principaux . . .



Implémentation de l'AsT

Un type Ast (type_ast.h)

```
typedef enum {OPERATION, VALEUR} TypeAst ;
typedef enum {N_PLUS, N_MUL, N_MOINS} TypeOperateur ;

typedef struct noeud {
        TypeAst nature ;
        TypeOperateur operateur ;
        struct noeud *gauche, *droite ;
        int valeur ;
} NoeudAst ;

typedef NoeudAst *Ast ;
```

Des primitives de construction (ast_construction.h)

```
Ast creer_operation(TypeOperateur opr , Ast op_gauche , Ast op_droit);
// renvoie un Ast (op_gauche, opr, op_droit) de nature OPERATION

Ast creer_valeur(int val) ;
// renvoie un Ast "feuille", de nature VALEUR et de valeur val
```

Implémentation de l'AsT

Un type Ast (type_ast.h)

```
typedef enum {OPERATION, VALEUR} TypeAst ;
typedef enum {N_PLUS, N_MUL, N_MOINS} TypeOperateur ;

typedef struct noeud {
        TypeAst nature ;
        TypeOperateur operateur ;
        struct noeud *gauche, *droite ;
        int valeur ;
} NoeudAst ;

typedef NoeudAst *Ast ;
```

Des primitives de construction (ast_construction.h)

```
Ast creer_operation(TypeOperateur opr , Ast op_gauche , Ast op_droit);
// renvoie un Ast (op_gauche, opr, op_droit) de nature OPERATION

Ast creer_valeur(int val) ;
// renvoie un Ast "feuille", de nature VALEUR et de valeur val
```

Construire un Ast lors de l'analyse

construction ascendante : feuilles $\rightarrow$ racine

$\Rightarrow$ Enrichir l'analyseur : ajouter des paramètres à chaque procédure

- le (sous) AsT résultat produit
- *parfois*, un sous AsT donné auxilliaire

```
Rec_facteur (resultat A : AsT) =
   selon LC.nature
        cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer
        cas PARO : Avancer ; Rec_eag (A) ;
                   si LC.nature = PARF alors Avancer sinon Erreur
        autre : Erreur
fin
```

Attention : en C les paramètres résultats sont des pointeurs !

```
*A = creer_valeur (LC.valeur)
```

Construire un Ast lors de l'analyse

construction ascendante : feuilles $\rightarrow$ racine

$\Rightarrow$ Enrichir l'analyseur : ajouter des paramètres à chaque procédure

- le (sous) AsT résultat produit
- *parfois*, un sous AsT donné auxilliaire

```
Rec_facteur (resultat A : AsT) =
   selon LC.nature
        cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer
        cas PARO : Avancer ; Rec_eag (A) ;
                   si LC.nature = PARF alors Avancer sinon Erreur
        autre : Erreur
fin
```

Attention : en C les paramètres résultats sont des pointeurs !

```
*A = creer_valeur (LC.valeur)
```

Construction de l'AsT (1)

$$eag \rightarrow seq\_terme$$

```
Rec_eag(A : resultat AsT) =
        Rec_seq_terme(A) ;
```

$$seq\_terme \rightarrow terme\ suite\_seq\_terme$$

```
Rec_seq_terme(A : resultat AsT) =
A1: Ast
Rec_terme(A1) ; Rec_suite_seq_terme(A1, A) ;
```

Construction de l'AsT (1)

eag $\rightarrow$ *seq_terme*

```
Rec_eag(A : resultat AsT) =
        Rec_seq_terme(A) ;
```

seq_terme $\rightarrow$ *terme suite_seq_terme*

```
Rec_seq_terme(A : resultat AsT) =
A1: Ast
Rec_terme(A1) ; Rec_suite_seq_terme(A1, A) ;
```

Construction de l'AsT (2)

suite_seq_terme $\rightarrow$ *op*1 *terme suite_seq_terme* [PLUS, MOINS]

suite_seq_terme $\rightarrow$ ε

```
Rec_suite_seq_terme(Ag : donnee Ast, A : resultat AsT) =
   Ad, A1: Ast
   Op : TypeOperateur
   selon LC().nature  // LC est le lexeme_courant()
      cas PLUS, MOINS :
               Rec_op1(Op) ; Rec_terme(Ad) ;
               A1 = creer_operation(Op, Ag, Ad) ;
               Rec_suite_seq_terme(A1, A) ;
      autre : A = Ag // Ne pas oublier !!!
fin
```

Idem pour Rec_seq_facteur() et Rec_suite_seq_facteur() ...

Construction de l'AsT (2)

suite_seq_terme $\rightarrow$ *op*1 *terme suite_seq_terme* [PLUS, MOINS]

suite_seq_terme $\rightarrow$ ε

```
Rec_suite_seq_terme(Ag : donnee Ast, A : resultat AsT) =
    Ad, A1: Ast
    Op : TypeOperateur
    selon LC().nature  // LC est le lexeme_courant()
       cas PLUS, MOINS :
                Rec_op1(Op) ; Rec_terme(Ad) ;
                A1 = creer_operation(Op, Ag, Ad) ;
                Rec_suite_seq_terme(A1, A) ;
       autre : A = Ag // Ne pas oublier !!!
fin
```

Idem pour Rec_seq_facteur() et Rec_suite_seq_facteur() ...

Construction de l'AsT (2)

suite_seq_terme $\rightarrow$ *op*1 *terme suite_seq_terme* [PLUS, MOINS]

suite_seq_terme $\rightarrow$ ε

```
Rec_suite_seq_terme(Ag : donnee Ast, A : resultat AsT) =
    Ad, A1: Ast
    Op : TypeOperateur
    selon LC().nature  // LC est le lexeme_courant()
       cas PLUS, MOINS :
                Rec_op1(Op) ; Rec_terme(Ad) ;
                A1 = creer_operation(Op, Ag, Ad) ;
                Rec_suite_seq_terme(A1, A) ;
       autre : A = Ag // Ne pas oublier !!!
fin
```

Idem pour Rec_seq_facteur() et Rec_suite_seq_facteur() ...

Construction de l'AsT (3)

facteur → *ENTIER*

facteur → PARO *eag* PARF

```
Rec_facteur (resultat A : AsT) =
   selon LC.nature
        cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer
        cas PARO : Avancer ; Rec_eag (A) ;
                   si LC.nature = PARF alors Avancer sinon Erreur
        autre : Erreur
fin
```

*op*1 → PLUS

*op*1 → MOINS

```
Rec_op1 (resultat Op : TypeOperateur) =
   selon LC.nature
        cas PLUS : Op := N_PLUS ; Avancer
        cas MOINS : Op := N_MOINS ; Avancer
        autre : Erreur
fin
```

Construction de l'AsT (3)

facteur → *ENTIER*

facteur → PARO *eag* PARF

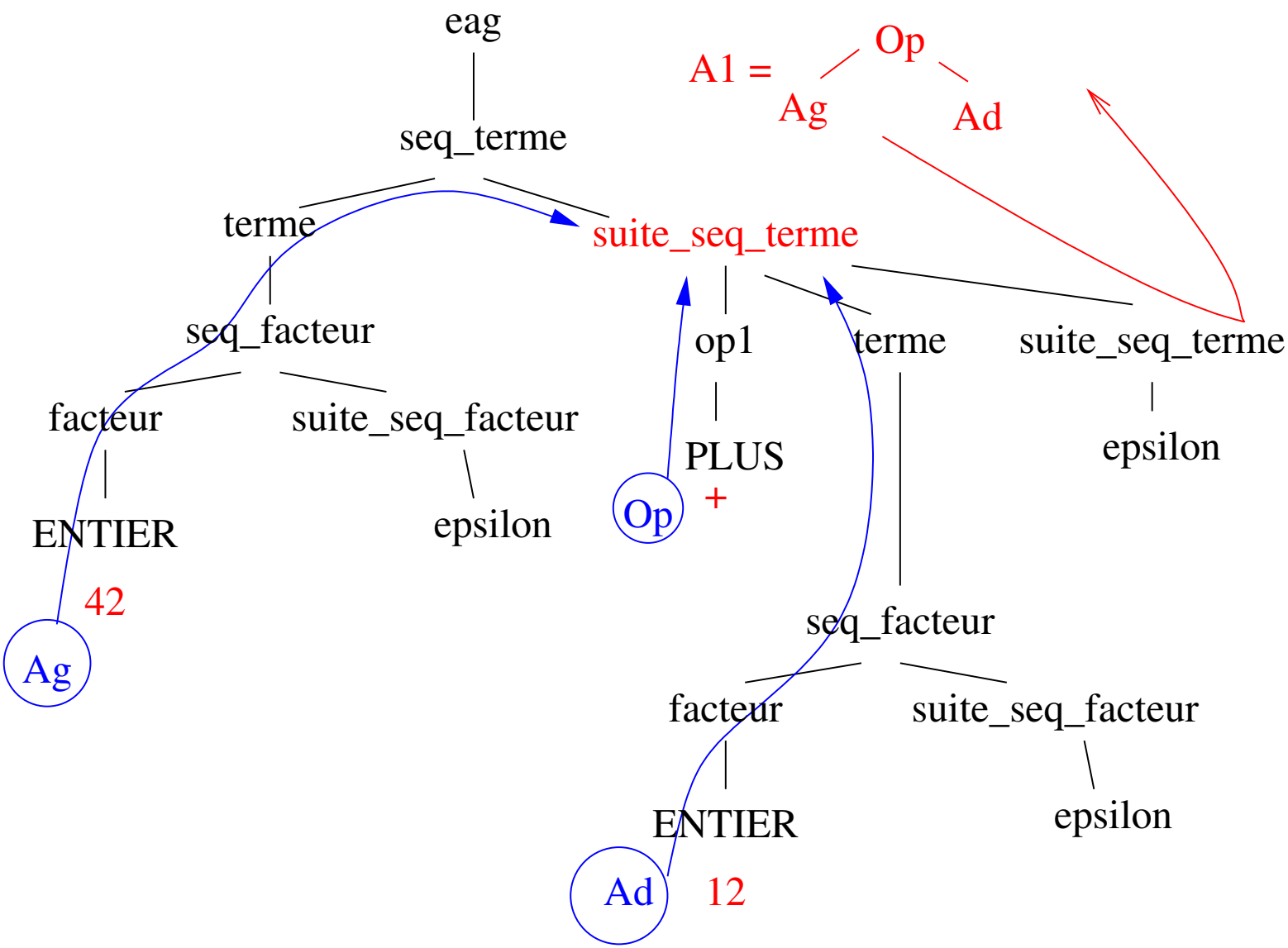
```
Rec_facteur (resultat A : AsT) =
   selon LC.nature
        cas ENTIER : A := creer_valeur (LC.valeur) ; Avancer
        cas PARO : Avancer ; Rec_eag (A) ;
                   si LC.nature = PARF alors Avancer sinon Erreur
        autre : Erreur
fin
```

*op*1 → PLUS

*op*1 → MOINS

```
Rec_op1 (resultat Op : TypeOperateur) =
   selon LC.nature
        cas PLUS : Op := N_PLUS ; Avancer
        cas MOINS : Op := N_MOINS ; Avancer
        autre : Erreur
fin
```

Exemple : 42 + 12



Calculer la valeur d'une EAG

Un (simple !) parcours **récursif** de l'AsT ...

```
fonction Evaluer (A : Ast) : entier
  Vg, Vd : entier
  selon A.Nature
      cas VALEUR : // feuille
          retourner A.valeur
      cas OPERATION  :   // operateur
          Vg = evaluer(A.fg) ;
          Vd = evaluer(A.fd) ;
          selon A.Operateur :
              cas N_PLUS : retourner Vg + Vd
              cas N_MOINS : ...
          fin
  fin
fin
```

Dans la suite (le dernier TP “calculette” !) . . .

1. **Finir** d'écrire l'analyseur syntaxique (sans construction de l'AsT) pour le langage *eag* à partir de la **Grammaire**.
2. Etendre cet analyseur syntaxique pour produire l'Ast.
3. Ecrire la fonction de calcul de la valeur d'un Ast
4. Intégrer le tout dans le programme principal . . .